# phy++ v1.0

# Contents

# Chapter 1

# Introduction

# Chapter 2

# Core library

## 2.1 Overview

At the core of the phy++ library is the *vector* class. This is basically an enhanced `std::vector`[1], and it therefore shares most of its features and strengths. In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *run-time*, meaning that its content can vary depending on user input, but also that a vector can change its total number of elements at any time. These elements are stored contiguously in memory, which provides optimal performances in most situations. Lastly, a vector is an homogeneous container, meaning that a given vector can only contain a single type of elements (e.g., `int` or `float`, but not both at the same time).

Like most advanced C++ libraries, phy++ is essentially *template* based. This means that most of the code is written to work for *any* type `T`, e.g., `int`, `float`, `std::string`, or whatever you need. However, while templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library. The good thing is, since phy++ is a numerical analysis library, we know in advance what types will most often be stored inside the vectors. So for this reason, to reduce typing and enhance readability, we introduce type aliases for the most commonly used vector types:

- `vec1f`: vector of `float`,
- `vec1d`: vector of `double`,
- `vec1i`: vector of `int` (precisely, `int_t = std::ptrdiff_t`),
- `vec1u`: vector of `unsigned int` (precisely, `uint_t = std::size_t`),

[1] In fact, `std::vector` is used to implement the phy++ vectors internally.

- `vec1b`: vector of `bool`,
- `vec1s`: vector of `std::string`,
- `vec1c`: vector of `char`.

On top of the `std::vector` interface, the phy++ vector adds some extra functionalities. The most important ones are operator overloading, multi-dimensional indexing, and array indexing.

### 2.1.1 Operator overloading

The only thing you can do to operate on all the elements of an `std::vector` is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. In phy++, we have *overloaded* the usual mathematical operators on our vector type, meaning that it is possible to write the above code in a much simpler way:

```
// Using phy++ vector.
vec1f w = {1,2,3,4};
w *= 2; // {2,4,6,8}
```

Not only this, but we can perform operations on a pair of vectors in the same way:

```
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y; // {5,5,5,5}
```

**Warning**

The only issue with operator overloading concerns the hat operator `^`. In most languages, this operator is used for exponentiation, e.g., `4^2 == 16`. However, in C++ this value is actually `6`, because the hat operator is the binary XOR (exclusive-or). Even worse, the hat operator in C++ does not have the same *precedence* as in regular mathematics: it has a lower priority than any other mathematical operator. Both these reasons make it unwise to overload the hat operator in `phy++`. In order to perform exponentiation, you will have to use a dedicated function such as `pow(4,2)` (**??**).

### 2.1.2 Multi-dimensional indexing

The standard `std::vector` is a purely linear container: one can access its elements using `v[i]`, with `i` ranging from `0` up to `std::vector::size()-1` (included). However, the `phy++` vector allows N-dimensional indexing, i.e., using a group of indices to identify one element. For example, this is particularly useful to work on images, which are essentially 2-dimensional objects where one identifies a given pixel by its coordinates `x` and `y`. The natural syntax for this indexing would be to write `img[x,y]`. This syntax is valid C++ code, but unfortunately will not do what you expect[2] and there is no sane way around it. The alternative we chose here is to write instead `img(x,y)`. While it is not as semantically clear as using brackets, it has the nice advantage of being compatible with the IDL syntax.

The multi-dimensional nature of a vector is determined at *compile time*, i.e., it cannot be changed after the vector is declared. By default, a vector is mono-dimensional. To use the above feature, one needs to specify the number of needed dimensions in the type of the vector. For example, a 2D image of `float` will be declared as `vec2f`. These type aliases are provided for dimensions up to 6. Here is an example of manipulation of a 2D matrix:

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};

// Index ordering is similar to C arrays: the last index
// is contiguous in memory. Note that this is *opposite*
// to the IDL convention.
m(0,0); // 1
m(0,1); // 2
m(1,0); // 4
```

[2]This will call the *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means `img[y]`. With proper configuration, most compiler will warn about this though, since in this context `x` is a useless statement, so you should be safe should you make this mistake.

```
// It is still possible to access elements as if in a
// "flat" vector
m[0]; // 1
m[1]; // 2
m[3]; // 4
```

**Advanced**

If for some reason you need to use more than 6 dimensions, or if you need to declare a vector of some type which is not covered above, you can always fall back to the full template syntax:

```
using vec12f = vec_t<12,float>;
using vec3cx = vec_t<3,std::complex>;
```

The hard limit on the number of dimensions will then depend on your compiler. The C++ standard does not guarantee anything, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a doctor first.
As for the types allowed inside the vector, there is no explicit restriction. However, some features may not be available depending on your type, and you will have to enable these yourself (operator overloading, in particular). Lastly, the phy++ vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

### 2.1.3 Array indexing

We have seen that, instead of accessing each element individually, we can use operator overloading to perform simple operations on all the elements of the vector at once: `w *= 2`. We can also, like with `std::vector`, modify each element individually, knowing their indices: `w[2] *= 2`. One last important feature allowed by the phy++ vector is array indexing, which allows us to create a *view* inside an array. Each element of the view is actually a reference to another element in the original array, and modifying the elements of the view actually modifies the values in the original array.

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the
// fifth elements. So we first create a vector containing
// the corresponding indices.
vec1u id = {1,2,4};

// Then we create the view.
// Note the usage of "auto" there. The type of the view is
// complex, and it is better not to worry about it.
auto v = w[id];

// Now we can modify these elements very simply, as if they
// were part of a real vector.
v;          // {2,3,5}
v *= 2;     // {4,6,10}
w;          // {1,4,6,4,10,6}
w[1] = 99; // {1,99,6,4,10,6}
v;          // {99,6,10}
```

**Warning**

It is important to note that, since a view keeps *references* to the elements of the original vector, its lifetime must not exceed that of the original vector. Else, it will contain *dangling* references that point to unused memory, and this should be avoided at all cost. In fact, views are not meant to be stored into named variables like in the above example. Most of the time, one will use them as temporary variables, e.g.:

```
vec1f w = {1,2,3,4,5,6};
vec1u id = {1,2,4};
w[id] *= 2; // {1,4,6,4,10,6}
```

## 2.2 The vector class

The full type of the vector class is

```
template<std::size_t Dimension, typename ElementType>
struct vec_t<Dimension,ElementType>;
```

In the rest of this document, `Dimension` will usually just be called `D` and `ElementType` will be shortened to `T`.

### 2.2.1 Member variables

A vector only contains two member variables:

- `std::vector<...> data;`
  This is the underlying `std::vector` containing the elements of the vector. It is exposed to the public interface for simplicity, but on most occasions it should *not* be used directly. In fact, it may become part of the private interface in the future, so you should not rely on its existence.

- `std::array<std::size_t,D> dims;`
  This variable contains the dimensions of this vector (useful only for multidimensional vectors). On no occasion should you modify this variable yourself: you should only read its content. To change the dimensions of a vector, either use `resize(...)` (2.2.3) or assign another vector (2.2.2).

  **Example**
  ```
  // Create a 2D image with 256 x 128 pixels
  vec2f img(256,128);
  img.dims[0]; // 256
  img.dims[1]; // 128
  for (std::size_t x = 0; x < img.dims[0]; ++x)
  for (std::size_t y = 0; y < img.dims[1]; ++y) {
      img(x,y) = 3.1415;
  }
  ```

### 2.2.2 Constructors and assignment

There are various ways to construct a new vector, or assign it some value. In the following, we will cover the various *constructors* and the associated *assignments* when applicable.

- `vec_t::vec_t(); // The default constructor`

  Like `std::vector`, the phy++ vector is *default constructible*. By default it is in a valid state where the vector does not contain any element.

  **Example**
  ```
  vec1f w; // default constructor: w is empty
  ```

- `explicit vec_t::vec_t(...); // The dimension constructor`

The `std::vector` has a constructor to create a new vector of a given size. The phy++ vector also offers this feature, however it is rendered a bit more complex by the possibility of having multi-dimensional vectors. In particular, it is possible to specify the size either by giving all the dimensions one by one, or by specifying some (or all) as a `std::array`. In all cases, the vector is populated with the number of requested elements, and all these elements are *default initialized* (i.e., integers and floats are initialized to `0`, booleans to `false`, and `std::string` are empty). This constructor is declared `explicit` to prevent interference with the other constructors and assignments.

**Example**

```
vec1f w(10);      // w contains 10 objects all equal to 0
vec2d z(5,4);     // z contains 5*4=20 objects all equal to 0
vec2s x(z.dims); // x has the same dimensions as z
                 // but contains strings, all empty
vec3b y(z.dims, 8); // y has the same dimensions as z,
                 // plus an extra dimension of length 8
```

- `vec_t::vec_t(nested_initializer_list<D,T>); // The list constructor`

Like `std::vector`, the phy++ vector can be initialized from a list of values. This usually requires the usage of `std::initializer_list<T>`, however here we also have to support multi-dimensional vectors, hence we need initializer lists of initializer lists of ..., and `nested_initializer_list<D,T>` is just that. Note that, since C++ is a row-major language, the most nested lists correspond to the last index in a multi-dimensional vector.

**Example**

```
vec1f w({1,2,3}); // 1D list constructor: {1,2,3}
vec2f z({{1,2,3},{6,5,4}}); // 2D list constructor: {{1,2,3},{6,5,4}}
z(0,0); // 1
z(0,1); // 2
z(0,2); // 3

// Assignment
w = {4,5,6};
z = {{1,2}, {3,4}, {5,6}}; // dimensions can change through assignment
z(0,0); // 1
z(1,0); // 3
z(2,0); // 5
```

- `vec_t::vec_t(const vec_t&); // The copy constructor`

  Like `std::vector`, the phy++ vector is *copiable*, meaning that one can duplicate the content of an existing vector inside another vector by copy. Note that this constructor is only valid for copying vectors of the *same* type. If the type is different, then another constructor is called (conversion constructor, see below).

  **Example**

  ```
  vec1f w = {1,2,3};
  vec1f z(w); // copy constructor: {1,2,3}

  // Assignment
  z = w;
  ```

- `vec_t::vec_t(vec_t&&); // The move constructor`

  Like `std::vector`, the phy++ vector is *movable*, meaning that one can move the content of an existing vector that is going to be destroyed inside another vector. This is an optimized copy for temporary variables (C++11 move semantics). You need not explicitly ask for either the copy of the move constructor, as they will automatically be chosen by the compiler. For you, this is transparent (but the performance boost is large).

  **Example**

  ```
  vec1f z(vec1f{1,2,3}); // move constructor: {1,2,3}
  // Here, a temporary vector is created with vec1f{1,2,3}.
  // This temporary vector is then *moved* inside z.

  // Assignment
  z = vec1f{4,5,6};
  ```

- `vec_t::vec_t(const vec_t<D,OtherT>&); // The conversion constructor`

  C++ supports implicit conversion between all the build in types. In particular, it is possible to write:

  ```
  int i = 0;
  float f = i; // int to float
  bool b1 = i; // int to bool, mostly for interoperability with C
  bool b2 = f; // float to bool, not sure if that makes sense...
  ```

While this is very convenient in most cases, it can also lead to dangerous silent conversions, such as the `float` to `bool` conversion. This is, somehow, a legacy of C. In phy++ we decided to also support such implicit conversions. They make the code much easier to read, but the price to pay is that sometimes we do some conversions which are not necessary, and we do not realize it because they are implicit. However, we decided to disable implicit conversion to and from `bool` vectors, since it could lead to subtle bugs that are difficult to trace. It is sill possible to do the conversion to `bool` using explicit

**Example**

```
vec1i w = {1,2,0};
vec1f z(w); // int to float {1,2,0}
vec1b y(w); // error: cannot convert int to bool
vec1s x(w); // error: cannot convert int to std::string

vec1b b = {true,true,false};
vec1i s(b); // bool to int {1,1,0}

// Assignment
z = w;
y = w;        // error: cannot convert int to bool
x = w;        // error: cannot convert int to std::string

s = b;        // error: implicit conversion is not allowed for bool
s = vec1i(b); // ok: bool to int {1,1,0}
```

- `vec_t::vec_t(const vec_t<D,T*>&); // The view constructor`

  The last constructor on the list is the view constructor. It allows implicit conversion of a view (2.3) into a new, independent vector.

**Example**

```
vec1i w = {1,2,3};
vec1u id = {0,1};
vec1i z(w[id]); // {1,2}

// Assignment
z = w[id];
```

### 2.2.3 Member functions

- `bool vec_t::empty() const;`

  This function will return **`true`** if this vector contains at least one element, and **`false`** otherwise. In particular, **`true`** will be returned for default constructed vectors, and after a call to `vec_t::clear()`.

  **Example**

  ```
  vec1i v;
  v.empty(); // true
  vec1i w = {1,2,3};
  w.empty(); // false
  ```

- `std::size_t vec_t::size() const;`

  This function will return the total number of elements in this vector. If the vector is empty, then the function returns `0`. If the vector is multidimensional, then the function returns the product of all the dimensions.

  **Example**

  ```
  vec1i v;
  v.size(); // 0
  vec1i w = {1,2,3};
  w.size(); // 3
  vec2i z = {{1,2,3}, {4,5,6}};
  z.size(); // 6
  ```

- `void vec_t::resize(...);`

  This function can be used to explicitly change the size of a vector. The parameters it accepts are the same as the dimension constructor (2.2.2), i.e., either integral values for individual dimensions, or an `std::array` containing multiple dimensions, or any combination of these. However, the total number of dimensions of the vector must remain unchanged.

  After the vector has been resized, its content will have changed. For a monodimensional vector, if the resize operation *decreased* the total number of elements, then the last elements will be erased, but the other ones will remain untouched. On the other hand, if the resize operation *increased* the total number of elements, all the previous elements are unchanged, and new elements are inserted at the end of the vector, default constructed (i.e., zeroes for integral types, etc.). For a multidimensional vector, its content is left in an *undefined state*, and can be assumed to be destroyed[3].

**Example**

```
vec1i v = {1,2,3};
v.resize(5); // {1,2,3,0,0}
v.resize(2); // {1,2}
v.resize(3); // {1,2,0}

vec2f w = {{1,2},{2,3}};
w.resize(2,3); // w has been resized, but its content is unspecified
w(0,0); // ?
```

- `void vec_t::clear();`

  This function removes all elements from the vector, and sets all dimensions to zero.

**Example**

```
vec1i v = {1,2,3};
v.clear(); // v is now empty
v.empty(); // true
v.size();  // 0
v.dims;    // {0}
```

- `T& vec_t::back();`

  This function is only available for monodimensional vectors. It returns the last element of the vector. Will crash if called on an empty vector.

**Example**

```
vec1i v = {1,2,3};
v.back(); // 3
v.back() == v[v.size()-1]; // always true
```

- `T& vec_t::front();`

[3]This is just out of laziness. In the future, this will probably be specified to behave like the monodimensional vector. The issue is that it involves non-trivial reshuffling of the elements to preserve the original structure after the dimensions have changed. I have not needed this feature so far.

This function is only available for monodimensional vectors. It returns the first element of the vector. Will crash if called on an empty vector.

**Example**

```
vec1i v = {1,2,3};
v.front(); // 1
v.front() == v[0]; // always true
```

- `void vec_t::push_back(...);`

  The behavior of this function is different for monodimensional and multidimensional vectors. For monodimensional vectors, this function appends a new element at the end of the vector, and therefore takes for argument a single value of type `T` (or convertible to `T`). For multidimensional vectors, this function takes for argument another vector of `D-1` dimensions, and whose lengths match the *last* `D-1` dimensions of the first vector. This new vector is inserted after the existing elements, and the first dimension of the first vector is increased by one.

  **Example**

  ```
  vec1i v = {1,2,3};
  v.push_back(4); // {1,2,3,4}

  vec2i w = {{1,2,3}, {4,5,6}};
  w.push_back({7,8,9}); // {{1,2,3}, {4,5,6}, {7,8,9}}
  w.push_back({7,8});   // error: dimensions do not match, 2 != 3
  ```

- `void vec_t::reserve(std::size_t);`

  This function is used for optimization, and is similar to `std::vector::reserve()` (actually, this function is called internally). To understand what this function actually does, one needs to know the internal behavior of `std::vector`. By default, the `std::vector` only allocates enough memory to hold a few elements. Once the allocated memory is full, `std::vector` allocates a larger amount of memory, copies the existing elements inside this new memory, and frees the old memory. This strategy allows virtually unlimited growth of a given vector, and is quite efficiently tuned. However, it is still an expensive operation, and performances can be greatly improved if one knows *in advance* the total number of objects that need to be stored in the vector, so that the right amount of memory is allocated from the beginning, and no copy is required. This function does just that, it tells `std::vector` how many elements it *will* (or might) contain at some point, so that the vector can already allocate enough memory. This is also useful if you only have a rough idea of the future number of elements.

**Example**

```
vec1i v = {1,2,3,...};

// Let's imagine you have an algorithm that will produce an
// unknown number of values, but you know that on average
// it is close to N*N.
vec1i w;
// Reserve roughly enough memory in advance
w.reserve(v.size()*v.size());

// Now the algorithm will run close to the optimal memory
// efficiency
for (...) {
    w.push_back(...);
}
```

- `vec1u vec_t::ids(std::size_t) const;`

  This is only useful for multidimensional vectors. This function converts a "flat" index into an array of multidimensional indices. The `vec_t::flat_id` function does the inverse job.

  **Example**

  ```
  vec2i v(2,3);
  v.ids(0); // {0,0}
  v.ids(1); // {0,1}
  v.ids(2); // {0,3}
  v.ids(3); // {1,0}
  v[3] == v(1,0); // true
  ```

- `std::size_t vec_t::flat_id(...) const;`

  This is only useful for multidimensional vectors. This function converts a group of multidimensional indices into a "flat" index. The `vec_t::ids` function does the inverse job.

  **Example**

```
vec2i v(2,3);
v.flat_id(0,0); // 0
v.flat_id(0,1); // 1
v.flat_id(0,2); // 2
v.flat_id(1,0); // 3
v(1,0) == v[3]; // true
```

- `std::size_t vec_t::pitch(std::size_t) const;`

  This is only useful for multidimensional vectors. This function returns the "pitch" factor associated to a given dimension. This factor is number of elements in memory that separate two consecutive indices of this dimension. By definition, the pitch factor is `1` for the last dimension. For the other dimensions, this is the product of all the other dimensions located between the one considered and the last dimension.

  **Example**

  ```
  vec3f v(5,8,6);
  v.pitch(2); // 1
  v.pitch(1); // 6 = v.dims[2]
  v.pitch(0); // 48 = v.dims[2]*v.dims[1]
  ```

- `bool vec_t::is_same(const vec_t<D,T>&) const;`

  This function tests if the provided vector is a view inside this vector.

  **Example**

  ```
  vec1f v = {1,2,3};
  vec1f w = {1,2,3};
  vec1u id = {1,2};
  v.is_same(v[id]); // true
  v.is_same(w[id]); // false
  ```

- `const vec_t& vec_t::concretize() const;`

  This function just returns a reference to this vector, and is only present to mirror the interface of the view class.

- `iterator vec_t::begin();` and `iterator vec_t::end();`

  These functions allow iteration over the values of this vector. The only reason one may use these functions explicitly is when using algorithms from the standard C++ library, which often work on a pair of iterators as returned by `begin()` and `end()`.

**Example**

```
vec1f v = {1,2,3};
// The presence of these functions allow to use vectors
// in range-based loops
for (float& f : v) {
    f += 1;
}
```

### 2.2.4 Indexing

There are two ways to index a given vector, for both mono- and multidimensional vectors. The first way is through “flat” indices and the bracket indexing `v[i]`, i.e., a single index that runs contiguously in memory, and the second way is through multidimensional indices and parenthesis indexing `v(i)` (2.1.2). For monodimensional vectors, these two methods are perfectly identical.

Flat indexing does not care about the details of the dimensions of a given vector. The only important thing is the total number of elements in the vector. This is the simplest and fastest[4] way to access the data inside a vector.

**Example**

```
vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.size(); ++i) {
    // Here we traverse the vector v regardless of its dimensions
    v[i] = 12.0 + i*i - sqrt(5.0*i) + v[i/4];
}
```

Multidimensional indexing is more involved computationally, because it implies some index arithmetic to compute the flat index and find the right place to read in memory. However it is much more expressive and easier to read and understand. Furthermore, except for very critical code sections, the performance gap is usually negligible, as long as you iterate over the dimensions properly, i.e., following the example below, that you do the *last* nested loop to iterate on the *last* index. This will guarantee as much memory locality as possible, and will take best advantage of CPU caches.

**Example**

[4] Actually there is a faster way using the `safe` wrapper, which does not do bounds checking. See the “Advanced” note at the end of this section.

```
vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.dims[0]; ++i)
for (std::size_t j = 0; j < v.dims[1]; ++j)
for (std::size_t k = 0; k < v.dims[2]; ++k) {
    // Here we traverse the vector v keeping its dimensional structure
    v(i,j,k) = 42.0 + i + j + k/(j+1);
}
```

Indexing a vector can only be done with integers, e.g., `int`, `unsigned int`, `int_t`, `uint_t`, `std::size_t`, etc. Indexing with *unsigned* integers is faster because it removes the need to check for the positivity of the index, and should therefore be preferred when possible. Negative indices are allowed though, and they are interpreted as *reverse* indices, where `-1` refers to the last element of the vector, `-2` the one before the last, etc.

**Example**

```
vec1i v = {1,2,3,4};
v[0];    // 1, int index
v[0u];   // 1, unsigned int index (faster, but more cumbersome)
vec[-1]; // 4, int index
vec[-2]; // 3, int index
vec[-1] == vec[vec.size()-1]; // always true

v[0.1]; // error: can only access using integers
```

As discussed in the overview (2.1.3), one can also use vector to index another one. This is called array indexing, by opposition to the scalar indexing we have seen just above. Similarly to scalar indexing, array indexing is only allowed by using integers, i.e., `vec1u`, `vec1i` and their multidimensional counterparts. Again, unsigned integers should be preferred when possible.

**Example**

```
vec1i v = {1,2,3,4};
vec1u id = {0,2,3};
v[id]; // 1,3,4
```

Sometimes, one will want to use array indexing to access all the elements at once, for example to set all the elements of a vector to a specific value. This can be done with a loop, of course, but it can be boring to write. To do so without explicitly writing the loop, one typically has to create first an index vector containing all the indices of the target vector, and then apply the operation:

```
vec1i v = {1,2,3,4};
```

```
vec1u id = {0,1,2,3}; // all the indices of v
v[id] = 12;

// Note that the following will not work
v = 12; // now v only contains a single element equal to 12
```

Not only is this not very practical to write, it is error prone and not very clear. If we decide to add an element to `v`, we also have to modify `id`. Not only this, but it will most likely be slower than writing the loop directly, because the compiler may not realize that you are accessing all the elements contiguously, and will fail to optimize it properly. For this reason, we also introduce the "placeholder" symbol, defined as a single underscore `_`. When used as an index, it means "all the indices in the range". Coming back to our example:

```
vec1i v = {1,2,3,4};
v[_] = 12; // it cannot get much shorter
```

This placeholder index can be used in all situations, with both flat and multidimensional indexing. It can be further refined to only encompass a fraction of the whole range, using a peculiar syntax[5]:

```
vec1i v = {1,2,3,4};
v[_-2] = 12;   // only access the indices 0 to 2
v[2-_] = 12;   // only access the indices 2 to 3
v[1-_-2] = 12; // only access the indices 1 to 2
```

### 2.2.5 Operators

The set of available operators depend on the type of the elements contained in the vector.

- For arithmetic types (`int`, `unsigned int`, `float` and `double`): addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and (integer types only) modulo (`%`).
- For `bool`: and (`&&`), or (`||`) and negation (`!`).
- For `std::string`: concatenation (`+`).
- For all types: less than (`<`), less than or equal (`<=`), greater than (`>`), greather than or equal (`>=`), equal (`==`) and not equal (`!=`).

[5]Be warned that this feature has been introduced recently and may not survive in the future.

In all cases, the operator overloading allows mixing together vectors of different types (as long as they are convertible one to another) and scalar values.

**Example**

```
vec1i v = {1,2,3,4};
v + 2; // {3,4,5,6}
v + v; // {2,4,6,8}

vec1s fruits = {"apple", "orange"};
"I ate an "+fruits; // {"I ate an apple", "I ate an orange"}
2+fruits;           // error: no operator found for int + std::string
```

## 2.3 The view class

The full type of the view class is

```
template<std::size_t Dimension, typename ElementType>
struct vec_t<Dimension,ElementType*>;
//              note the asterisk ^
```

The public interface of the view class is very similar to that of the normal vector, and we will not repeat it here. There are some important differences though, which are inherent to the goal of this class. In particular, there is no available constructor (you do now create a view yourself, you ask for it from an existing vector that will create it for you), and the `resize()` function is not available. Lastly, the view implements `concretize()`, differently.

Thanks to their strong similarity, we will not distinguish in the other sections between vectors and views, and will consider views as just another kind of vectors. Indeed, the interface of these two classes has been designed for views to be completely interchangeable with vectors, and vice versa, so that the code of any given function is generally written once and is valid for both types.

### 2.3.1 Member functions

- `vec_t<D,T> vec_t<D,T*>::concretize() const`

  This function creates a new vector out of the elements of this view. The returned vector is completely independent from this view, or the original vector this view is currently pointing to. This function is mostly useful when writing generic functions. Indeed, views are implicitly convertible to normal vectors on assignment (2.2.2).

## 2.4 Metaprogramming traits and functions (WIP)

**Warning**

This whole section is more advanced than the rest. It is describing the sets of helper types and metaprogramming functions that one can use to write new functions. It is intended to be followed by readers already familiar with the `phy++` library, and with good knowledge of C++ metaprogramming.

# Chapter 3

# Support libraries

In this chapter we describe the set of helper functions that are part of the `phy++` support library. These functions are not essential to the use of the `phy++` library, but are mostly modular components that one may choose to use or not. All the support functions are sorted into broad categories to help you discover new functions and algorithm. Alternatively, if you know the name of a function and would like to read its documentation, an index is available at the end of this document (4).

## 3.1 Generic vector functions

The vector and view classes are useful and complete tools. However, there are a number of tasks that one repeatedly need to do, like generating a sequence of indices, or sorting a vector, and that would be tedious to write each time they are needed. In this section we list these functions and algorithms.

-
  ```
  vec_t<D,int_t>  indgen(...)
  vec_t<D,uint_t> uindgen(...)
  vec_t<D,float>  findgen(...)
  vec_t<D,double> dindgen(...)
  ```

  These functions will create a new vector, whose dimensions are specified in argument (like in the dimension constructor, 2.2.2, or `vec_t::resize()`, 2.2.3). After this vector is created, the function will fill it with values that start at `0` and increment by steps of `1` until the end of the vector.

**Example**

```
vec1i v = indgen(5);     // {0,1,2,3,4}
vec2u w = uindgen(3,2); // {{0,1}, {2,3}, {4,5}}
```

- `vec1u where(const vec_t<D,bool>&)`

  This function will scan the `bool` vector provided in argument, will store the flat indices (2.2.4) of each element which is `true`, and will return all these indices in a vector. This is a very useful tool to filter and selectively modify vectors, and probably one of the most used function of the whole library.

  **Example**

  ```
  vec1i v = {4,8,6,7,5,2,3,9,0};
  // We want to select all the elements which are greater than 3
  // We use where() to get their indices
  vec1u id = where(v > 3); // {0,1,2,3,4,7}
  // Now we can check
  v[id]; // {4,8,6,7,5,9}, good!

  // The argument of where() can by as complex as you want
  id = where(v < 3 || (v > 3 && v % 6 < 2)); // now guess

  // It can also involve multiple vectors, as long as they have
  // the same dimensions
  vec1i w = {9,8,6,1,-2,0,8,5,1};
  id = where(v > w || (v + w) % 5 == 0);
  // The returned indices are valid for both v and w
  v[id]; // {8,6,7,5,2,9}
  w[id]; // {8,6,1,-2,0,5}
  ```

- `complement(const vec_t& v, const vec1u& id)`

  This function works in tandem with `where`. Given a vector `v` and a set of indices `id`, it will return the complementary set of indices inside this vector, i.e., all the indices of `v` that are *not* present in `id`.

  **Example**

  ```
  vec1i v = {1,5,6,3,7};
  vec1u id = where(v > 4); // {1,2,4}
  vec1u cid = complement(v, id); // {0,3}
  ```

- `vec1u sort(const vec_t&)`

  `void inplace_sort(vec_t&)`

  These functions will change the order of the elements inside a given vector so that they are sorted from the smallest to the largest. The difference between the two versions is that `sort` does not actually modify the provided vector, but rather returns a vector containing indices inside the provided vector, and `inplace_sort` directly modifies the provided vector. The later is the fastest of the two, but it is less powerful.

  **Example**

  ```
  // First version
  vec1i v = {1,5,6,3,7};
  vec1u id = sort(v); // {0,3,1,2,4}
  v[id]; // {1,3,5,6,7} is sorted
  // now, id can also be used to modify the order of
  // another vector of the same dimensions

  // Second version
  inplace_sort(v);
  v; // {1,3,5,6,7} is sorted
  ```

- `vec1u uniq(const vec_t& v)`

  `vec1u uniq(const vec_t& v, const vec1u& sid)`

  This function will traverse the provided vector `v` and find all the unique values. It will store the indices of these values (if a value is present more than once inside `v`, the index of the first one will be used) and return them inside an index vector. The first version assumes that the values in `v` are *sorted* from the smallest to the largest. In the second version, `v` may not be sorted, but the second argument `id` contains indices that will sort `v` (e.g., `id` can be the return value of `sort(v)`).

  **Example**

```
// For a sorted vector
vec1i v = {1,1,2,5,5,6,9,9,10};
vec1u u = uniq(v); // {0,2,3,5,6,8}
v[u]; // {1,2,5,6,9,10} only unique values

// For an non-sorted vector
vec1i w = {5,6,7,8,6,5,4,1,2,5};
vec1u u = uniq(w, sort(w));
w[u]; // {1,2,4,5,6,7,8}
```

- `vec_t<1,T> flatten(const vec_t<D,T>&)`

  This function transforms a multidimensional vector into a monodimensional vector. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. The `reform` function does the inverse job.

  **Example**

  ```
  vec2i v = {{1,2,3}, {4,5,6}};
  vec1i w = flatten(v); // {1,2,3,4,5,6}
  ```

- `vec_t<D,T> reform(const vec_t<1,T>&)`

  This function transforms a monodimensional vector into a multidimensional vector. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. However, the provided dimensions have to generate the same number of elements as the provided vector contains. The `flatten` function does the inverse job.

  **Example**

  ```
  vec1i v = {1,2,3,4,5,6};
  vec2i w = reform(v, 2, 3); // {{1,2,3}, {4,5,6}}
  ```

- `vec_t<1,T> reverse(const vec_t<1,T>&)`

  This function will inverse the order of all the elements inside the provided vector.

  **Example**

  ```
  vec1i v = {1,2,3,4,5,6};
  vec1i w = reverse(v); // {6,5,4,3,2,1}
  ```

- `vec_t<N,T>   replicate(T, ...)`

  `vec_t<D+N,T> replicate(const vec_t<D,T>&, ...)`

  This function will take the provided scalar (first version) or vector (second version), and replicate it multiple times according to the provided additional parameters, to generate additional dimensions.

  **Example**

  ```
  // First version
  vec1i v = replicate(2, 5); // {2,2,2,2,2} 5 times 2
  vec2i w = replicate(2, 3, 2); // {{2,2},{2,2},{2,2}} 3 x 2 times 2

  // Second version
  vec2i z = replicate(vec1i{1,2}, 3); // {{1,2},{1,2},{1,2}} 3 times {1,2}
  // Note that it is not possible to just use a plain initializer list
  vec2i z = replicate({1,2}, 3); // error, unfortunately
  ```

- `vec_t<1,T> remove(const vec_t<1,T>& v, const vec1u& id)`

  `void inplace_remove(vec_t<1,T>& v, const vec1u& id)`

  These functions will remove the elements in `v` that have their indices in `id`. The only difference between the first and the second version is that the former will first make a copy of the provided vector, remove elements inside this copy, and return it. The second version modifies directly the provided vector, and is therefore faster.

  **Example**

  ```
  // First version
  vec1i v = {4,5,2,8,1};
  vec1i w = remove(v, {1,3}); // {4,2,1}

  // Second version
  inplace_remove(v, {1,3});
  v; // {4,2,1}
  ```

- `void append<N>(vec_t<D,T1>& t1, const vec_t<D,T2>& t2)`

  `void append(vec_t<1,T1>& t1, const vec_t<1,T2>& t2)`

  `void prepend<N>(vec_t<D,T1>& t1, const vec_t<D,T2>& t2)`

  `void prepend(vec_t<1,T1>& t1, const vec_t<1,T2>& t2)`

These functions behave similarly to `vec_t::push_back`, in that they will add new elements at the end (`append`), but also at the beginning (`prepend`) of the provided vector. However, when `vec_t::push_back` can only add new elements from a vector that is one dimension less than the original vector (or a scalar, for monodimensional vectors), these functions will add new elements from a vector of the *same* dimension. These functions are also more powerful than `vec_t::push_back`, because they allow you to choose along which dimension the new elements will be added, through the template parameter `N` (note that this parameter is useless and therefore does not exist for monodimensional vectors). The other dimensions must be identical. They are clearly dedicated to a more advanced usage.

**Example**

```
// For monodimensional vectors
vec1i v = {1,2,3};
vec1i w = {4,5,6};
append(v, w);
v; // {1,2,3,4,5,6}
prepend(v, w);
v; // {4,5,6,1,2,3,4,5,6}

// For multidimensional vectors
vec2i x = {{1,2}, {3,4}};          // x is (2x2)
vec2i y = {{0}, {0}};              // y is (1x2)
vec2i z = {{5,6,7}};               // z is (3x1)
append<1>(x, y);
x; // {{1,2,0}, {3,4,0}}           // x is (2x3)
prepend<0>(x, z);
x; // {{5,6,7}, {1,2,0}, {3,4,0}} // x is (3x3)
```

- `vec_t<1,T> shift(const vec_t<1,T>& v, int_t n, T d = 0)`

  This function will shift the position of the elements inside the provided vector `v` by a given amount of indices `n`. Elements that would go outside of the bounds of the vector are destroyed. New elements are inserted and default constructed, or assigned the default value `d` (optional argument).

  **Example**

```
vec1i v = {1,2,3,4,5};
vec1i sr = shift(v, 2);
sr; // {0,0,1,2,3}
vec1i sl = shift(v, -2, 99);
sl; // {3,4,5,99,99};
```

- `bool is_any_of(T1 v1, const vec_t<D2,T2>& v2)`

  `vec_t<D1,bool> is_any_of(const vec_t<D1,T1>& v1, const vec_t<D2,T2>& v2)`

  This function looks if there is any value inside `v2` that is equal to `v1`. If so, it returns `true`, else it returns `false`. The second version is just the vectorized version of the first, where `is_any_of` is called for each element inside `v1`.

  **Example**

  ```
  vec1i v = {7,4,2,1,6};
  vec1i d = {5,6,7};
  vec1b b = is_any_of(v, d); // {true, false, false, false, true}
  ```

- `void match(const vec_t& v1, const vec_t& v2, vec1u& id1, vec1u& id2)`

  This function traverses `v1` and, for each value in `v1`, looks for elements in `v2` that have the same value. If one is found, the flat index of the element of `v1` is added to `id1`, and the flat index of the element of `v2` is added to `id2`. Then the function goes on to the next value in `v1`. Note that, contrary to `match_dictionary`, each value in `v1` is matched to *at most* one value in `v2`, and vice versa. In other words, if `v1` or `v2` contain duplicates, only the first value will be matched and the others will be ignored. This function is symmetric: the result will be the same if you swap the two input vectors (of course, the output vectors have to be swapped also).

  **Example**

  ```
  vec1i v = {7,4,2,1,6};
  vec1i w = {2,6,5,3};
  vec1u id1, id2;
  match(v, w, id1, id2);
  id1; // {2,4}
  id2; // {0,1}
  v[id1] == w[id2]; // always true
  ```

- `void match_dictionary(const vec_t& v1, const vec_t& v2, vec1u& id1, vec1u& id2)`

This function traverses `v1` and, for each value in `v1`, looks for an element in `v2` that has the same value. If one is found, the flat index of the element of `v1` is added to `id1`, and the flat index of the element of `v2` is added to `id2`. Then the function goes on to the next value in `v1`. Contrary to `match`, each value in `v2` can be matched to multiple values in `v1`. Therefore this function is not symmetric: the second vector should be considered as a "dictionary", hence the name of this function. It is assumed that `v2` does not contain any duplicates.

**Example**

```
vec1i v = {7,6,2,1,6};
vec1i w = {2,6,5,3};
vec1u id1, id2;
match_dictionary(v, w, id1, id2);
id1; // {1,2,4}
id2; // {1,0,1}
v[id1] == w[id2]; // always true
```

- `void phypp_check(bool b, ...)`

  This function makes error checking easier. When called, it checks the value of `b`. If `b` is **`true`**, then nothing happens. However if `b` is **`false`**, then the current file and line are printed to the standard output, followed by the other arguments of this function (they are supposed to be an error message explaining what went wrong), and an assertion is raised, immediately stopping the program. This function is used everywhere in the `phy++` library to make sure that certain conditions are properly satisfied before doing a calculation, and it is essential to make the program crash in case something is unexpected (rather than letting it run hoping for the best, and often getting the worst).

  Since this function is actually implemented by a preprocessor macro, you should not worry about its performance impact. It will only affect the performances when something goes wrong and the program is about to crash. However the calculation of `b` can itself be costly (for example, you may want to check that a vector is sorted), and there is no way around this.

  **Example**

```
vec1i v;
// Suppose v is read from the command line arguments.
v = read_from_somewhere();

// The following code needs at least 3 elements in the
// vector v, so we need to check that first.
phypp_check(v.size() >= 3, "this algorithm needs at least "
    "3 values in the input vector, but only ", v.size(),
    " were found");

// If we get past this point, then v has the right number
// of elements, and we can proceed
do_stuff(v);
```

## 3.2 Parsing command line arguments

- `void read_args(int argc, char* argv[], ...)`

## 3.3 File input/output

- `bool file::exists(std::string)`
- `bool file::is_older(std::string f1, std::string f2)`
- `vec1s file::list_directories(std::string)`
- `vec1s file::list_files(std::string)`
- `std::string file::directorize(std::string)`
- `std::string file::get_basename(std::string)`
- `std::string file::get_directory(std::string)`
- `bool file::mkdir(std::string)`
- `void file::read_table(std::string f, uint_t s, ...)`
- `void file::write_table(std::string f, uint_t w, ...)`
  `void file::write_table_csv(std::string f, uint_t w, ...)`
  `void file::write_table_hdr(std::string f, uint_t w, vec1s hdr, ...)`

## 3.4 String manipulation

- `std::string strn(T)`

  `std::string strn(T v, uint_t n, char fill = '0')`

  `std::string strn_sci(T v)`

- `vec_<D,std::string> strna(const vec_t<D,T>&)`

  `vec_<D,std::string> strna(const vec_t<D,T>& v, uint_t n, char fill = '0')`

  `vec_<D,std::string> strna_sci(const vec_t<D,T>& v)`

- `bool from_string(std::string s, T& v)`

- `bool empty(std::string s)`

- `uint_t length(std::string s)`

- `std::string trim(std::string s, std::string cs)`

- `std::string toupper(std::string s)`

- `std::string tolower(std::string s)`

- `std::string replace(std::string s, std::string p, std::string r)`

- `vec1s split(std::string s, std::string p)`

- `vec1s cut(std::string s, uint_t n)`

- `vec1s wrap(std::string s, uint_t w, std::string i = "", bool e = false)`

- `std::string collapse(vec_t<D,std::string>)`

  `std::string collapse(vec_t<D,std::string> v, std::string s)`

- `uint_t find(std::string s, std::string p)`

- `bool match(std::string s, std::string r)`

  `bool match_any_of(std::string s, vec1s r)`

- `bool start_with(std::string s, std::string p)`

  `bool end_with(std::string s, std::string p)`

- `std::string erase_begin(std::string s, std::string p)`
  `std::string erase_begin(std::string s, uint_t n)`
  `std::string erase_end(std::string s, std::string p)`
  `std::string erase_end(std::string s, uint_t n)`
- `std::string keep_start(std::string s, uint_t n = 1)`
  `std::string keep_end(std::string s, uint_t n = 1)`
- `std::string remove_extension(std::string)`
- `std::string align_left(std::string s, uint_t w, char f = ' ')`
  `std::string align_center(std::string s, uint_t w, char f = ' ')`
  `std::string align_right(std::string s, uint_t w, char f = ' ')`
- `uint_t distance(std::string s1, std::string s2)`

## 3.5 OS interaction

- `std::string system_var(std::string v, std::string d = "")`

## 3.6 Printing to the terminal

- `void print(...)`
  `void error(...)`
  `void warning(...)`
  `void note(...)`

## 3.7 Measuring time

- `double now()`
- `std::string today()`
- `std::string time_str(double)`
- `std::string seconds_str(double)`

- `auto progress_start(uint_t)`

  `void progress(auto p, uint_t m)`

  `void print_progress(auto p, T i, uint_t m)`

## 3.8 Math and generic algorithms

## 3.9 Parallel execution

- `void fork(std::string)`

  `void spawn(std::string)`

- `auto thread::pool(uint_t)`

- `void thread::sleep_for(double)`

## 3.10 FITS input/output

## 3.11 Image processing

## 3.12 Astrophysics

### 3.12.1 Catalog management

### 3.12.2 Position cross-matching

### 3.12.3 Image stacking

### 3.12.4 Template fitting

# Chapter 4

# Function index

# Index